# 3

# FROM RM-ODP TO THE FORMAL BEHAVIOR REPRESENTATION

**Pavel Balabko, Alain Wegmann**

*Laboratory of Systemic Modeling*
*Ecole Polytechnique Fédérale de Lausanne*
*EPFL-IC-LAMS*
*{pavel.balabko,alain.wegmann}@epfl.ch*

**Abstract**. In this work we consider the behavioral aspects of system modeling. In order to specify the behavior of a system, many different notations can be used. Quite often, different terms in these notations are related to the same element in a system implementation. In order to relate these terms and guarantee the consistency between different notations, a standard framework should be used. In this work we show how the Reference Model for Open Distributed Processing (RM-ODP) can be used for the purpose of the mapping of terms from different behavioral notations. RM-ODP behavior models are based on the concept of Time Specific Action. Time Specific Actions represent directly things that happen in the Universe of Discourse with explicit reference to time. However the explicit reference to time leads to a considerable loss of abstractness. To elevate the level of abstraction we have considered Time Abstracted RM-ODP models where concrete time information is omitted. We used Time Abstracted RM-ODP models to show the correspondence between terms in UML Activity Diagrams, UML Statechart Diagrams and CCS process algebra by means of relating them with RM-ODP terms. This allows us to consider RM-ODP as a possible meta-model for behavior specifications written in UML. It can help to insure the consistency of UML models.

## 1 INTRODUCTION

Behavior models play a central role in system specifications. Many specification languages can be used to specify the behavior of a business and IT systems. A system designer chooses a particular language depending on the designer's experience and on the problems he is trying to solve. For example, to show the conformance of the implementation of a system behavior with its specification, a system designer can use formal languages (for example, Pi-calculus). To visualize the state machine of a developed system, a system designer may use a UML statechart diagram or activity diagram (a variation of a state machine in which the states represent the performance of actions or subactivities [O1999]). The design of complex systems requires that a system designer solve many problems simultaneously (visualize a model, check the conformance of a model, etcetera), thus several specification languages should be used. This raises a problem: a system designer needs to build several independent models of the same system. This leads to

the duplication of the information, which can be an additional source of errors: models done in different languages can be inconsistent.

To avoid building several mutually dependent models, we can build a generic model (see Figure 1).

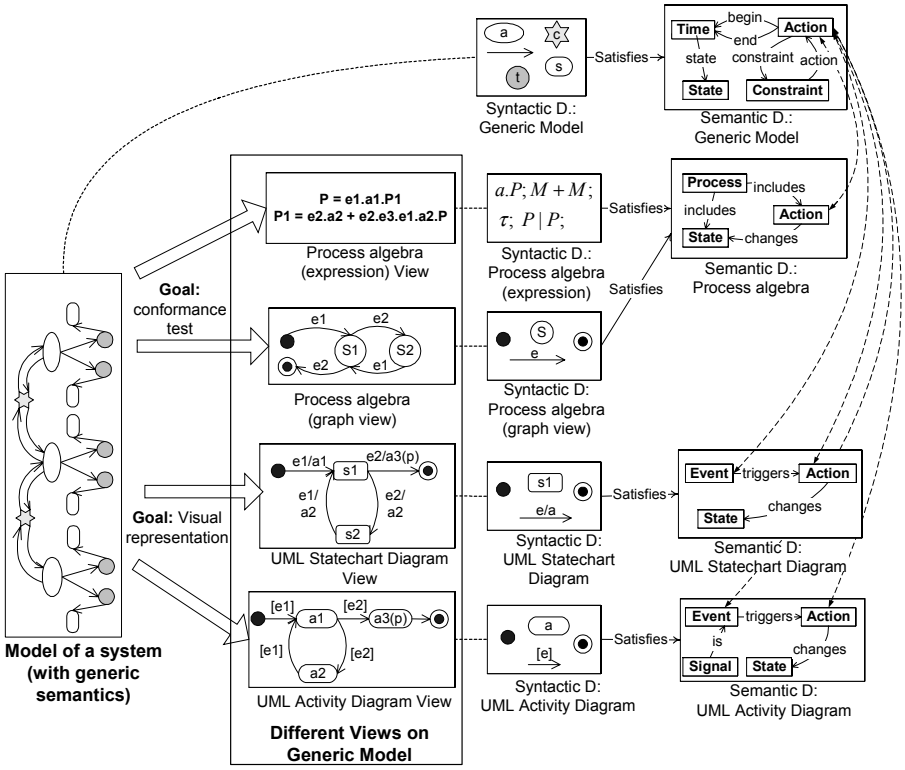



**Figure 1**. Generic Model and different views on a generic model

All other models can be considered as views of this generic model. Any view on a model should address some particular problems that a system designer wants to solve. Any view is based on a particular specification language. J. Wing in [W1990] defines specification language as a triple <*Syntax, Semantics, Satisfies*>, where *Syntax* is called a language syntactic domain; *Semantics* is a language semantic domain, and *Satisfies* is "satisfies relation". *Satisfies relation* defines the relation between syntactic terms in a system specification and their semantic meanings in a semantic domain.

The semantic domain of the generic model should cover the semantics of all possible views that a generic model can have. In other words, any concept in the semantic domain of any view should be mapped with one or more concepts in the semantic domain of the generic model. This shows that the semantic domain of the generic model should be generic enough to include all fundamental concepts for the specification of business and IT systems.

In this work we propose to use the Part 2: Foundations of the Open Distributed Processing - Reference Model RM-ODP [I1996] as a semantic domain for the

generic model. "RM-ODP, ITU-T Recommendations X.901 to X.904 | ISO/IEC 10746, is based on precise concepts derived from current distributed processing developments" [I1996]. We choose RM-ODP because "RM-ODP introduces generic terms that apply to any form of modeling activity" [N2001] and RM-ODP provides rigorous semantics for these terms. We use a formalization of this semantics written in the Alloy language[i]. This formalization was proposed in [N2001], where Naumenko shows the classification of RM-ODP concepts with the aid of the set theory and using regular predicate logic. RM-ODP concepts described in [N2001] can be used only as a basis for the semantics of the generic behavior model. The work of Naumenko contains too many different concepts: not all of them are related with behavior modeling. Thus in this work we consider only a subset of concepts from [N2000] and refine some of them in order to define semantics for generic behavior models.

In section 2 we consider the minimum set of RM-ODP concepts that we need to build generic behavior models. We define more precisely some RM-ODP behavior modeling concepts (particularly behavioral constraints, time and state). We have to do this because RM-ODP does not define all modeling concepts precisely enough to relate them with other existing formal notations. One of the basic RM-ODP modeling concepts that we consider in section 2 is action. It represents directly things that happen in the Universe of Discourse with explicit reference to time. In other words, any action (or time specific action) is specified for the particular time interval. We call a model built with time specific actions Time Specific RM-ODP model.

Time Specific RM-ODP model can not be used to specify an infinite behavior that may contain infinitely many actions. To specify infinite behavior, a system designer has to use action types. In section 3 we introduce two action types: Time Abstracted Action (section 3.1) and Parameterized Time Abstracted Action (section 3.1). Based on these two types, we define a Time Abstracted RM-ODP model. This model can be taken as a generic from figure 1. The main contribution of this chapter consists in making explicit the relations between Time Specific and Time Independent RM-ODP models.

In section 4 we show an example of how a Time Abstracted RM-ODP model can be used as a generic model. We show how it can be seen from the three views done with the following specification languages: CCS process algebra, UML Activity Diagram and UML Statechart Diagram. Section 5 is a conclusion.

## 2 RM-ODP A GENERIC SEMANTIC DOMAIN

In this section we consider the concepts from the RM-ODP semantic domain that are necessary for the modeling of the behavior of systems.

The basic concepts that we use in our work are taken from the clause 8 "Basic modeling concepts" of the RM-ODP Part 2. These concepts are: action, time, and state. According to [N2001] these concepts are essentially the first-order propositions about model elements. We will also use some concepts (type, instance, precondition,

postcondition) from the clause 9 “Specification concepts”. Specification concepts are the higher-order propositions applied to the first-order propositions about the model elements. Wegmann [W2001] states: **“***Basic Modeling Concepts* and generic *Specification Concepts* are defined by RM-ODP as two independent conceptual categories. Essentially, they are two qualitative dimensions that are necessary for defining model elements that correspond to entities from the universe of discourse”.

To explain the semantics of the generic model more clearly, we will use the Alloy formalism. Alloy is a simple modeling language that allows a modeler to describe the conceptual space of a problem domain. Using Alloy we specify the RM-ODP semantic domain.

RM-ODP conceptual elements from the semantic domain can be partitioned in the following way:

```
model RM-ODP {
domain {ODP_Concepts}
state {
partition … BasicModellingConcepts, SpecificationConcepts : static ODP_Concepts
…
}
```

**Code Fragment 1.** RM-ODP model

Let’s consider the minimum set of modeling concepts (Basic Modeling Concepts and Specification Concepts) necessary for the specification of systems behavior. There are a number of approaches for specifying the behavior of distributed systems coming from people with different backgrounds and considering different aspects of behavior. “However, they can almost all be described in terms of a single formal model” [L1990]. Based on Lamport, to specify the behavior of a concurrent system a system designer has “to specify a set of states, a set of action and a set of behavior”. Each behavior is modeled as a finite or infinite sequence of interchangeable states and actions. To describe this sequence there are mainly two dual approaches. According to [B1991] they are:

1. “Modeling systems by describing their set of actions and their behaviors”.
2. “Modeling systems by describing their state spaces and their possible sequences of state changes”.

“These views are dual in the sense that an action can be understood to define state changes, and state changes occurring in state sequences can be understood as abstract representations of actions” [B1991]. In our work we consider both of these approaches as an abstraction of the more general approach based on RM-ODP. In the next subsection we consider the first approach where we give the definition of action and behavior. Then we consider the definition of state and state structure. Finally we show how state and behavior are related, thus showing their duality.

## 2.1 Action Structure

In this subsection we show how systems are specified “by describing their set of actions and their behaviors”. Action in RM-ODP is defined as:

***Action: “****Something which happens”.*

This definition means that "action characterizes a model element for its being "something that happens" [W2001]. To specify a model element as an action we have to consider two other modeling concepts that model changes happening in a system when an action occur. They are *state* and *time*. The definition of the state concept is given in the next subsection. The concept of time is a fundamental concept in modeling of systems. Based on RM-ODP time is a basic modeling concept that is used to specify the beginning and the end of an action[ii]. Therefore each RM-ODP action is bound to the specific time interval. That is why in our work we call RM-ODP action as *Time Specific Action (TSAction):*

```
partition …, TSAction, Time, … : static BasicModellingConcepts
                                 // Time and TSAction are BasicModellingConcepts

instant_begin : TSAction –>Time!        // each TSAction has one time point when it starts
instant_end : TSAction –>Time!          // each TSAction has one time point when it finishes
```

**Code Fragment 2.** Beginning and end of TSAction

However RM-ODP does not explain how time is modeled. A system designer has to decide how accurate he wants to model time. Henri Poincaré in [P1983] shows that a precise clock that can be used for time measurement does not exist in practice but only in theory. So the measurement of the time is always approximate. In this case we should not choose the most precise clocks, but those that explain the investigated phenomena in the best way. "Simultaneity of two events or their sequentiality, equality of two durations should be defined in the way that the formulation of the physical laws is the easiest" [P1983]. According to this idea we can choose different models of time. RM-ODP confirms this idea by saying that "a location in space or time is defined relative to some suitable coordinate system" [clause 8.10]. The time coordinate system defines a clock used for system modeling.

In our work we consider a time coordinate system as a partially ordered set of time points. Each point can be used to specify the beginning or the end of TSAction. A time coordinate system must have the following fundamental properties:

- Time is always increasing. This means that sequences of time points can not have loops.
- Any time point is defined in relation to other time points (next, previous or not related). This corresponds to the partial order defined on the set of time points.

We use the following formalization of time in Alloy: time is defined as a set of time points. Any time point has to be defined in relation with some other time points (partial order):

```
nextTE: Time -> Time        // defines the set of nearest following time points for any time point
                            // note that any time point may include several nextTE time points
```

We will also use the followingTE relation to define the set of the following time points or the transitive closure of the time point t over the nextTE relation:

```
// part of Alloy time declaration
followingTE: Time ->Time      // defines all possible following time points
```

Using followingTE we can write the following Alloy invariant[iii] that defines the transitive closure and guarantees that time point sequences do not have loops:

```
inv TimeInvariant {
all t: Time  |                                          // For all time points t
((no t.nextTE)->(no t.followingTE)) &&                  // (if t does not have nextTE it also does not have
                                                        followingTE) and
 ((some t.nextTE && no t.nextTE.followingTE)            // (if t has the nextTE that does not have any
                                                            followingTE
  ->(t.followingTE=t.nextTE )) &&                       //  then t.followingTE is equal to t.nextTE) and
  ((some t.nextTE && some t.nextTE.followingTE)         //  (if t has the nextTE that has some followingTE
  ->(t.followingTE=t.nextTE.followingTE + t.nextTE )) && //   then t.followingTE includes t.nextTE
                                                            and t.nextTE.followingTE) and
(t not in t.followingTE)                                // (time does not have loops)
}
```

**Code Fragment 3.** Time invariant

Now, using the already defined concept of *Time* we can give a formal Alloy definition of *TSAction*:

```
def TSAction{
  all a: TSAction                       // for each TSAction a
  | some t1:a.instant_begin             // [ (exists t1 = a.instant_begin) and
  | some t2: a.instant_end              // (exists t2 = a.instant_end) ] then
  | (t2 in t1.followingTE)              // (t2 happens after t1)
}
```

**Code Fragment 4.** TSAction

In this definition we suppose that the duration of any TSAction is not equal to zero (t1 can not be equal to t2). But in certain cases we can make an abstraction of the information about the fact that TSAction starts and ends in different time points (to define so called instantaneous actions). For this purpose we have to use an abstraction of time information that we consider in section 3.

To make a specification that includes more than one TSAction, we have to consider how TSActions in a specification can be structured. We use the RM-ODP *behavior* concept to define the TSAction structure:

***Behavior: "****A collection of [Time Specific] Actions with a set of [Time Specific Behavioral] Constraints on when they may occur",*

That can be formally represented in the following way:

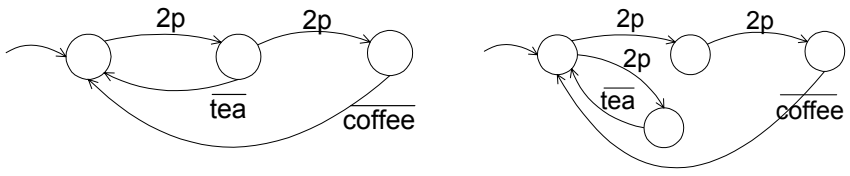
```
// part of Alloy behavior declaration
Behavior: BasicModellingConcepts
partition TSAction, TSBehavioralConstraints: static Behavior
          // Behavior is partitioned into the set of actions and the set of constraints.
corresponding_constraint (~constrained_action) : TSAction -> TSBehavioralConstraints
          // TSActions defined with corresponding TSBehavioralConstraints and vice versa.

def Behavior {
all b: Behavior |          // For any element b from Behavior set; (note that behavior is
                           // partitioned into the set of TSActions and the set of
                           // TSBehavioralConstraints)
  ( (b in TSAction) &&                      // [ (if b is a TSAction) then
  (some b.corresponding_constraint) ))  ||  //   (b has a at least one corresponding_constraint) ]
                                                and
  ( (b in TSBehavioralConstraints) &&       // [ (if b is a TSBehavioralConstraint) then
  (some b.constrained_action) )             //   (b has a at least one constrained_action) ]
}
```

**Code Fragment 5.** Behavior

This definition uses a concept called *(TimeSpecificBehavioral) constraint*. RM-ODP does not give us the precise definition of these constraints. But it gives some examples. Constraints may include, for example, constraints of sequentiality, non-determinism, concurrency or real-time constraints. From the definition of behavior, we can only conclude that TSBehavioralConstraints are part of a system behavior and that they are associated with TSActions (see the formal definition above). We will extend the definition of behavioral constraints in the next subsection.

### *2.1.1 Time Specific Behavioral Constraints*

Many modeling techniques represent behavioral constraints implicitly. Quite often we can infer them from behavior representation, like a transition graph. For example, figure 2 shows an example from the Milner's book [M1999] with two different specifications of a coffee/tea vending machine. This machine accepts coins of value 2p and provides a customer with coffee or tea. To get a coffee or tea a customer has to introduce coins and press a corresponding button (coffee or tea). The price for tea is 2p and the price for coffee is 4p. Figure 2.a shows the specification that has only constraints of sequentiality, since in any state of a system the next action is precisely defined depending on the request of a customer. Figure 2.b shows the specification with constraints of sequentiality as well as constraints of non-determinism. We can infer that the system in figure 2.b is specified using constraints of non-determinism; "after we have put in the first 2p, it may be in a state in which we can only get tea (it will not accept a further 2p), or it may be in a state in which we can only put in more money to get coffee" [M1999]. These two specifications "are annoyingly different for a thirsty user"




**(a)** **(b)**

**Figure 2.** Specification of the system using: a - sequential deterministic constraints; b - sequential and non-deterministic constraints

We base our approach on RM-ODP, where BehavioralConstraints are represented explicitly ("Behaviour of an object: A collection of (TS) Actions with a set of (TS) Behavioral Constraints on when they may occur" [I1996]). In our work we show how TSBehavioralConstraints can be made explicit: how the behavior of a system can be specified using a set of TSAction and TSBehavioralConstraints of sequentiality and non-determinism.

*Constraints of Sequentiality*

We begin with the analysis of TSBehavioralConstraints of sequentiality (TSSeqConstraints in Alloy code fragment 6). Each TSSeqConstraint of sequentiality should have the following properties:

- It is defined between two or more TSActions.
- Sequentiality has to guarantee that one TSAction is finished before the next one begins.

```
TSSeqConstraints: TSBehavioralConstraints                    // TSSeqConstraints are
TSBehavioralConstraints

def TSSeqConstraints {
all sc: TSSeqConstraints |                  // for any sc: TSSeqConstraints
some a1, a2: TSAction | (a1 != a2) &&       // (there are two different TSActions a1, a2) such that
    (a1 in sc.constrained_action) && (a2 in sc.constrained_action) &&   // (sc is defined between
                                                                          a1 and a2) and
   ( (a2.instant_begin in a1.instant_end.followingTE) ||   // [ (a1 is before a2) or
   (a1.instant_begin in a2.instant_end.followingTE) )      //   (a2 is before a1) ]
}
```

**Code Fragment 6.** TSBehavioralConstraints of Sequentiality

The Alloy definition from the code fragment 6 requires TSSeqConstraints to have a minimum of two sequential actions that happen one after another. But this Alloy definition does not tell us which TSActions happen first. To specify this we use two Alloy relations (seq_constraint and next_actions) and SeqInvariant (see code fragment 7). The seq_constraint relation relates a given TSAction (let's call it *tsa)* to one TSSeqConstraint. Then the next_actions relation relates TSSeqConstraint to the set of the TSActions. This set of action is the set of next TSActions for *tsa*.

```
seq_constraint: TSAction->TSSeqConstraints!  // for any TSAction there is one TSSeqConstraint
                                               that connect TSAction with next TSActions
next_actions: TSSeqConstraints -> TSAction   // any TSSeqConstraint can have several next
                                               TSActions
inv SeqInvariant {
  all sc:SeqConstraints |                      // for any sequential constraints sc and
     all a1:sc.constrained_action |            // for all TSActions a1 and a2
     all a2:sc.constrained_action |            // constrained by sc

    (  (a2.instant_begin in
            a1.instant_end.followingTE) ->     //   if a1 is before a2 then
             ( (sc=a1.seq_constraint) &&       // [ (sc is seq_constraint for a1) &&
               (a2 in sc.next_actions) &&      //   (sc includes a2 as the next action) &&
               (a1 not in sc.next_actions) &&  //   (sc does not include a1 as the next action) &&
               (sc not in a2.seq_constraint)   ) //   (sc is not sequential constraint for a2) ]
     ) &&                                      //   AND
     (  (a1.instant_begin in
             a2.instant_end.followingTE) ->    //  if a2 is before a1 then
               ( (sc=a2.seq_constraint) &&     // [ (sc is seq_constraint for a2) &&
                 (a1 in sc.next_actions)  &&   //   ((sc includes a1 as the next action) &&
                 (a2 not in sc.next_actions) &&  //   (sc does not include a2 as the next action) &&
                 (sc not in a1.seq_constraint)  ) //   (sc is not sequential constraint for a1) ]

    )
}
```

**Code Fragment 7.** Invariant that defines the sequence of TSActions

To illustrate the Alloy definition of TSSeqConstraints we show the example of the model (see figure 3) that corresponds to the formal Alloy semantics given above.

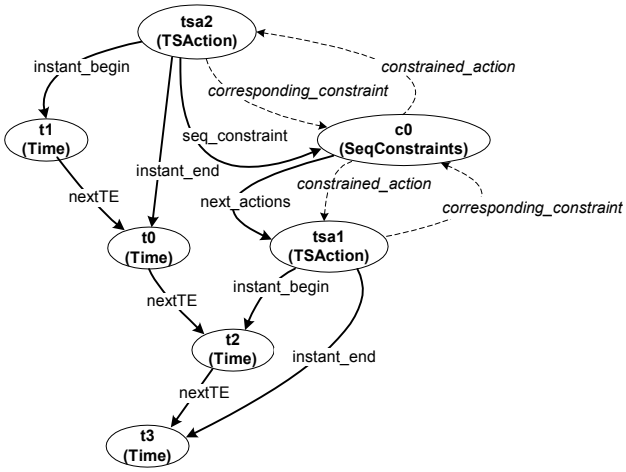



**Figure 3.** Example of the model of a system behavior built with Alloy Constraint Analyzer

The model from figure 3 was built with the Alloy Constraint Analyzer[iv]. This model is a result of the analysis of formal behavior semantics done with alloy Constraint Analyzer. The Alloy Constraint Analyzer checks the consistency of the formal semantics, randomly generates a sample configuration and visualizes it. Figure 3 shows a model that consists of the set of TSActions {tsa1, tsa2}, the set of TSBehavioralConstraints {c0}, the set of time points {t1, t0, t2, t3} and relations between model elements. Note that labels for model elements in figure 3 are generated automatically. That is why these labels are not ordered. We can see that the constraint c0 is the TSBehavioralConstraint of sequentiality between two TSActions tsa1 and tsa2. In figure 3 we show the corresponding_constraint and constrained_action relations with dotted arrows. We do it because these two relations do not do not have a particular interest for us for the rest of this work. They have been used only to define TS behavioral constraints. Thus we do not show these relations in following figures. Instead we use the seq_constraint and next_actions relations to show the sequence of TSActions.

The fact that the Alloy Constraint Analyzer has found a sample model allows us to conclude that formal behavioral semantics done in Alloy does not contain contradictions.

The definition of the constraints of sequentiality allows us to specify the semantics of the concepts defined in the section 13 of RM-ODP "Activity[v] Structure". Here we give two examples (for Chain of actions and Head action) that show how the formal semantics for these two concepts can be done based on the constraints of sequentiality.

***Head action:*** *In a given activity, an action that has no predecessor.*

```
def HeadAction{
 all ha:HeadAction|                          // for all ha:HeadAction
     no a:TSAction|                          // does not exist any a:TSAction
     ha in a.seq_constraint.next_actions     // such that ha is successor of a
}
```

Additionally we have to guarantee that all TSActions that do not have predeccessors are Head actions. We do it with the following Alloy invariant:

```
inv HeadActionInvariant {
  all a:TSAction| (no a1:TSAction| a in a1.seq_constraint.next_actions) ->(a in HeadAction)
}
```

Another concept that can be formalized using constraints of sequentiality is a chain of actions:

***Chain (of actions):*** *A sequence of actions within an activity where, for each adjacent pair of actions, occurrence of the first action is necessary for the occurrence of the second action.*

Based on the definition of synthesis constraints, we have to require that for any action in a chain maximum one successor and maximum one predecessor is possible:

```
def Chain {
 all ch:Chain |                                          // for all chains of actions
     (not sole ch.actions_in_chain ) &&                  // {there are min 2 action} &&
     ( all a:ch.actions_in_chain |                       // {for all actions a in chain ch:
      (   ( one a1: ch.actions_in_chain |                //  [ (there is one
           a in a1.seq_constraint.next_actions) ||       //     predecessor action a1) or
         ( a in HeadAction )  ) &&                       //     (a is Head action) ] &&
     ( one a.seq_constraint.next_actions ||              //  [ (there is one successor) or
       no a.seq_constraint.next_actions ) &&             //     (there is no successors) ] &&
     ( one a2: ch.actions_in_chain | a2 in HeadAction )  //  [one Head action per chain]}
     )
}
```

*Constraints of Non-determinism*

In order to formalize TSBehavioralConstraints of non-determinism we considered the following definition given in [B1991]: "A system is called non-deterministic if it is likely to have shown a number of different behaviors, where the choice of the behavior cannot be influenced by its environment". This definition of non-deterministic constraints is given from the point of view of the external observer of a system: when the external observer can not predict the reaction of a system after an interaction with a system. This means that the system at one point makes an internal choice between a minimum of two "branches" of different behavior.

Let's see how this definition works for the example from figure 2.b. In figure 2.b we can see that when a user of the coffee machine introduces first *2p*, the system can enter into two different states and therefore it can have two different behaviors: it will wait for the second *2p* or will provide *tea* for the user of the coffee machine. Thus a system has two different behaviors and the choice of the behavior can not be influenced by its environment.

In a general form, TSBehavioralConstraints of non-determinism should be defined between a minimum of three TSActions. The first TSAction should precede the two following internal TSActions. We can write this in Alloy in the following way:

```
TSNonDetermConstraints: TSBehavioralConstraints      // TSSeqConstraints are
TSBehavioralConstraints

def TSNonDetermConstraints {
all ndc: TSNonDetermConstraints |                    // for any ndc:
TSNonDetermConstraints
   some a1:TSAction |                                // (there is an TSAction a1) and
   some a2, a3 in  InternalTSAction |                // (there are two internal TSActions a2
   and a3) such that
```

```
   (a1 in ndc.constrained_action) &&                    // (sc is defined for a1) and
   (a2 in ndc.constrained_action) &&                    // (sc is defined for a2) and
   (a3 in ndc.constrained_action) &&                    // (sc is defined for a3) and
   (a2.instant_begin in a1.instant_end.followingTE) &&  // (a1 is before a2) and
   (a3.instant_begin in a1.instant_end.followingTE)     // (a1 is before a3)
}
```

**Code Fragment 8.** Constraints of non-determinism

Note that intuitively we may think to model a constraint of non-determinism as an internal action that makes a non-deterministic choice between two (or more) following actions. Can we really do that? An action that makes a choice between two "branches" of behavior should be specified with two (or more) different post-states.[vi] Each post-state defines a separate "branch" of behavior. But in our case we use time specific actions. This means that each action has a particular time when it starts and ends. As we will show in the next section, each time moment is associated to only one state. Thus the specification of a non-deterministic choice is not possible using TSAction and we use behavioral constraints to represent it in our models.

The discussion from the previous paragraph shows that the semantics of behavioral concepts would not be complete without considering the state of an object: "an object is characterized by its behavior and, dually, by its state" [I1996]. In the next section we discuss the definition of the state of an object and relate the concept of state with behavioral concepts considered above.

## 2.2 State Structure

Here we consider the second approach based on "Modeling systems by describing their state spaces and their possible sequences of state changes" [B1991]. We begin with RM-ODP definition of *state*:

***[TS]State (of an object) (RM-ODP, Part 2, clause 8.7):*** *At a given instant in time, the condition of an object that determines the set of all sequences of [TS]Actions in which the object can take part.*

This definition shows that the state of an object is defined in a given time point. That is why we call this state as Time Specific State (TSState).

In this work we use some simplifications. Since in this paper we consider the behavior only for one object, we do not make objects explicit on diagrams and in Alloy code. Therefore we declare TSState in Alloy without making a reference to an object:

```
// part of Alloy state declaration
state-existence: Time! -> TSState_!          // state is defined at a given moment in time
```

This Alloy definition taken from [N2001] can hardly be used in practice: to make specifications of complex systems it is not enough to specify TSState of an object in any point in time. We have to specify particular details that show how the TSState of an object changes. For this purpose we use the state structure:

***TSState Structure (of an object):*** *A set of attributes, a set of attribute values.*

Based on the TSState Structure we can specify states of each attribute. The state of an attribute specify the value that this attribute has in a given time point. Each

action can change values of some attributes while keeping other attributes unchanged. The composition of states of all attributes of an object gives us the composite state:

***Composite TSState (of an object)**: Composition of states of all attributes of an object.*

To specify the Composite TSState of an object we will use a function that specifies the relation between attributes and their values at a given moment in time. In Alloy this definition is written in the following way:

```
// part of Alloy declarations
partition … Information … : static BasicModellingConcepts     // Information is a basic modeling
                                                                 concept
partition StructuralInfo, BehavioralInfo : static Information  // Information can be structural
                                                                 and behavioral
// State Structure
Attrs, AVals: StructuralInfo          // state structure: set of attributes and attribute values
attrValue [Time]: Attrs -> AVals!     // any attribute has one value at a given moment
```

**Code Fragment 9.** Structural and behavioral information

Note that our definition of a Composite TSState extends the definition of the state proposed in RM-ODP. A Composite TSState shows how RM-ODP state can be specified as a composition of the states of several attributes.

As we said above "an action can be understood to define state changes and state changes occurring in state sequences can be understood as abstract representations of actions" [B1991]. This shows that TSState is dual with the concept of TSAction and these modeling concepts cannot be considered separately. To show the duality of TSAction and TSState we have to extend the definition of TSAction from the previous subsection in order to show that TSActions changes the state of a system:

```
def TSAction{
  all a: TSAction                                  // for each TSAction a
          | some attr: Attrs                       // there is at least one attribute such that
          | some t1:a.instant_begin                // (if t1 = a.instant_begin) and
          | some t2: a.instant_end                 // (if t2 = a.instant_begin) then
          | (t2 in t1.followingTE) &&              // [(t2 happens after t1) and
          (attr.attrState[t1] != attr.attrState[t2] ) // (attributes change their values in this
TSAction)]
}
```

**Code Fragment 10.** TSAction (new definition)

Note that in this definition each TSAction changes the value of at least one attribute. To understand it, let's go back to the definition of state. To determine the sequence of TSActions in which an object can take part, TSState has to keep information about which TSActions are already executed, which TSActions are currently executed, and which TSActions can be executed in the future. Thus each TSAction changes at least one attribute in the TSState of an object. This attribute keeps information about the fact that this TSAction is finished (or not)[vii].

Figure 4 shows the example of the model of state structure corresponding to the Alloy formal semantics.

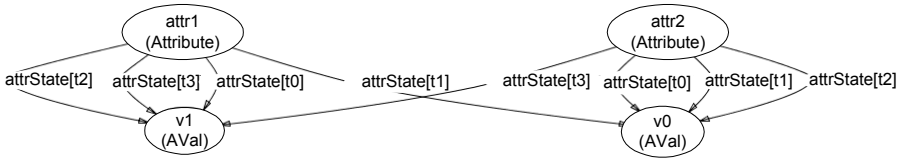



**Figure 4.** Example of the model of a system state, built with Alloy Constraint Analyzer

This example continues the example from figure 3. It shows that a system has two attributes (attr1, attr2). Each attribute may have two values (v1, v0) in different time points (t0, t1, t2, t3). By analyzing two diagrams from figure 3 and figure 4 we can see that the TSAction tsa2 changes the value of the attribute attr1 (*attr1.attrState[t1]=v0* and *attr2.attrState[t0]=v1*) and the TSAction tsa1 changes the value of the attribute attr2 (*attr2.attrState[t2]=v0* and *attr2.attrState[t3]=v1*).

## 2.3 Example of Complete Time Specific RM-ODP Model

The semantics of RM-ODP makes explicit how TSState and TSAction structures are related to each other. But the visual representation of models provided by Alloy Constraint Analyzer is not yet explicit enough. It represents TSState and TSAction structures separately (see figure 5 and figure 6). However, figure 5 and figure 6 are related by means of time points: any TSAction is defined between two time points (see code fragment 9) and any TSState is defined for a given time point (see code fragment 8). In order to explicit this relation between TSAction and TSState structures and to simplify Alloy diagrams, we use our notation. We use ovals to represent TSActions, rounded rectangles to represent TSStates. Each TSState is specified as a composition of TSStates of systems attributes. To represent time points we use small gray circles and to represent behavioral constraints we use stars. To represent relations between model elements we use arrows named in the same way as in figures 3 and 4 with a slight difference. First, we do not show *corresponding_constraint* and *constrained_action* relations. We show only seq_constraint and next_actions relations that we use to indicate the sequence of actions. Second, instead of showing states of each attribute in a given time point, we show the state of all attributes together. For this purpose we use state existence relation. In our work we call diagrams built using this notation Time Specific RM-ODP diagrams. Figure 5 shows an example of such diagram that corresponds to the model automatically generated with Alloy Constraint Analyzer. This example is based on the models from figures 3 and 4: the specification of the states of attributes from figure 4 was added to the specification of behavior from figure 3. The states of attr1 and attr2 are shown as parts of the composite states.

Figure 3
Figure 4

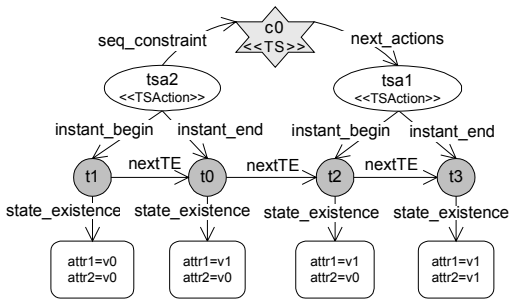



**Figure 5.** Time Specific RM-ODP model that combines the state structure from figure 5 and the TSAction structure from figure 6

We call a model specified with TSActions, a *Time Specific RM-ODP model*. As we can see a Time Specific RM-ODP model is precise but quite bulky (it contains too many details), even if the behavior to be modeled is simple. Fortunately, we can use a number of abstractions and simplifications to reduce the complexity of the model. Using simplifications can bring us to other different models. Further in our work we show some simplifications that can bring us to some existing modeling techniques: CCS process algebra, UML Statecharts and UML Activity diagrams.

# 3 TIME ABSTRACTED AND PARAMETRIC RM-ODP MODEL

As we have seen in section 2, Time Specific RM-ODP models have precise semantics that explain how different RM-ODP model elements are related to each other. However Time Specific RM-ODP models can not be used for modeling of the behavior with infinitely many TSActions. The behavior of an object may contain infinitely many TSActions due to the two following reasons. First, if the specification of the behavior is not limited in time. In this case, the sequence of actions would be unlimited. In order to make a finite specification of the infinite sequence of actions, we have to make an abstraction of time. In section 3.1 we show how an abstraction of time can be done. Second, the specification of behavior may contain infinitely many actions if at some point in time only one TSAction is possible out of the infinitely many TSActions. For example, if an object receives from its environment a single value out of infinitely many possible values, then using Time Specific RM-ODP model we have to specify a separate TSAction for each possible value. We have to do this because each TSAction can have only one post-state that would correspond to the reception of a concrete value. This will result in infinitely many TSActions and states of an object. In section 3.2 we show how to deal with this problem by means of specifying parameterized actions.

## 3.1 Time Abstracted Actions

System designers often do not make explicit time information and keep only constraints of sequentiality. Sometimes the presence of time information makes modeling precise, however "the incorporation of concrete timing properties leads to a considerable loss of abstractness" [B1991]. For example, using only TSAction does not allow specifying infinite behavior since it requires infinite sequence of TSActions. To make the specification of infinite behavior, we have to consider an abstraction of actual time information.

Based on the definition of TSAction, any TSAction changes the values of some attributes. We have also mentioned in section 2 that any TSAction must change the value of at least one attribute. This attribute or attributes show the state of a TSAction (if this TSAction has been finished or not). We call attributes that show the state of a TSAction, *temporal attributes*. These attributes specify which TSActions can be executed next. Hence we call them temporal. All other attributes we will call *ordinary attributes*. In Alloy code we partition the set of all attributes to the set of temporal attributes and the set of ordinary attributes.

```
partition TAttrs, OAttrs :static Attrs          // attrubutes can be temporal or ordinary
```

For example, figure 6 shows the example from the previous section where we distinguish between temporal and ordinary attributes.

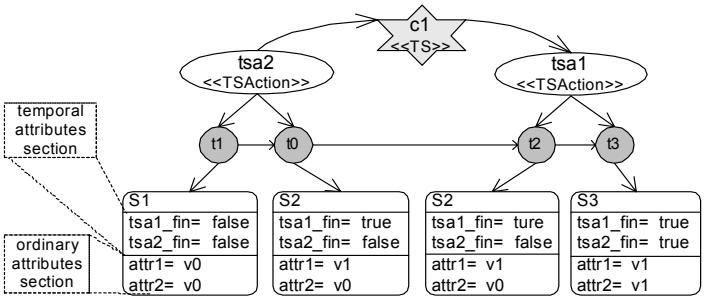



**Figure 6.** Time Specific RM-ODP model

Now we can define a predicate that characterizes the collection of TSActions that have the same result. For this purpose we use specification concepts presented in section 2. Among specification concepts we use pre- and post-conditions[viii]. In order to define a collection of TSActions with the same result, we will use TAPreconditions and TAPostconditions:

***TAPrecondition:*** *precondition in the form: equals(attr,val) (or "attr = val"), where attr∈{ordinary attributes} and val∈{values of ordinary attributes}.*

***TAPostcondition:*** *postcondition in the form: equals(attr,val) (or "attr = val"), where attr∈{ordinary attributes} and val∈{values of ordinary attributes}.*

```
pre_attributes: TAPrecondition -> OAttrs!    // one precondition specifies the value of one OAttrs
post_attributes: TAPostcondition -> OAttrs! // one postcondition specifies the value of one OAttrs
pre_values: TAPrecondition -> AVals!  // preconditions includes values for the ordinary attributes
post_values: TAPostcondition -> AVals!     // postconditions includes values for the ordinary
                                                attributes
```

Using pre and postconditions we can define a type of TSAction that we call: *Time Abstracted Action (TAAction).* We define it in the following way:

***Type of TSAction (TAAction):*** *It is a type characterizing the set of TSActions: it specifies values of all ordinary attributes before and after any TSAction (that is an instance of TAAction). These values are specified with TAPreconditions and TAPostconditions.*

***Instance of TAAction ():*** *TSAction that satisfies TAAction.*

Note that the definition of TAAction is a predicate stating that each TAAction requires ordinary attributes to have certain values before and after TSActions. Let's consider how this predicate can be expressed in Alloy.

```
…TAAction, TAPreconditions, TAPostconditions…: SpecificationConcepts
satisfies_type(~type_for): TSAction -> TAAction+        // each TSAction has at least one type
TAA_preconditions: TAAction -> TAPreconditions+         // each TAAction has at least one
                                                           TAPrecondition
TAA_postconditions: TAAction -> TAPostconditions+       // each TAAction has at least one
                                                           TAPrecondition
```

Based on the definition of TAAction we have to require that each TAAction has TAPrecondition and TAPostcondition for each ordinary attributes (OAttrs):

```
def  TAAction{
  all taa:TAAction |
    all tsa:TAAction.type_for |                  // for all instancess of TAAction: tsa (TSAction)
    all attr:OAttrs |                            // and for any ordinary attribute attr
    one pre: taa.TAA_preconditions|              // there is one precondition for taa
    one post: taa.TAA_postconditions |           // there is one postconditions for taa,   such that
    attr = pre.pre_attribute  &&                 // (attr is an attribute of the pre precondition) &&
    attr = post.post_attribute &&                // (attr is an attribute of the post postcondition) &&
   (one t:tsa.instant_begin |  attr.attrValue[t] = pre.pre_value) &&
                               // (the pre precondition specify the value of attr before TAAction) &&
   (one t:tsa.instant_end |  attr.attrValue[t] = post.post_value)
                              // (value of attrs is the same as the value of the precondtion)
 }
```

The class of TSAction is defined in the following way:

***Class of TSActions****: A set of TSAction satisfying a TAAction type.*

To define formally the Class of TSActions, for each class we have to indicate which TSActions should be included in this class. In Alloy we can do this in the following way:

```
… TAAction_Class…: SpecificationConcepts          // TAAction_Class is a specification concept
associated_type: TAAction_Class!->TAAction!          // TAAction_Class has a corresponding
                                                        type (TAAction)
member_of(~members): TSAction->TAAction_Class+    // each TSAction belongs to at least one
                                                     TAAction_Class
def TAAction_Class{
all c:TAAction_Class |                            // for every TAAction_Class
c.associated_type in c.members.satisfies_type     // the type for the TAAction_Class is the
                                                     same as the type for members of this
                                                     class
}
```

To better illustrate these definitions we use the example from section 2. In section 2 the example was used to show a sample model generated by Alloy Constraint Analyzer. This model included two time specific actions (tsa1 and tsa2) and two attributes (attr1 and attr2). Here we show other elements of the model that we did not show in section 2. These elements are specification concepts: TAActions, TAAction_Classes, TAPreconditions and TAPostconditions (See figure 7).

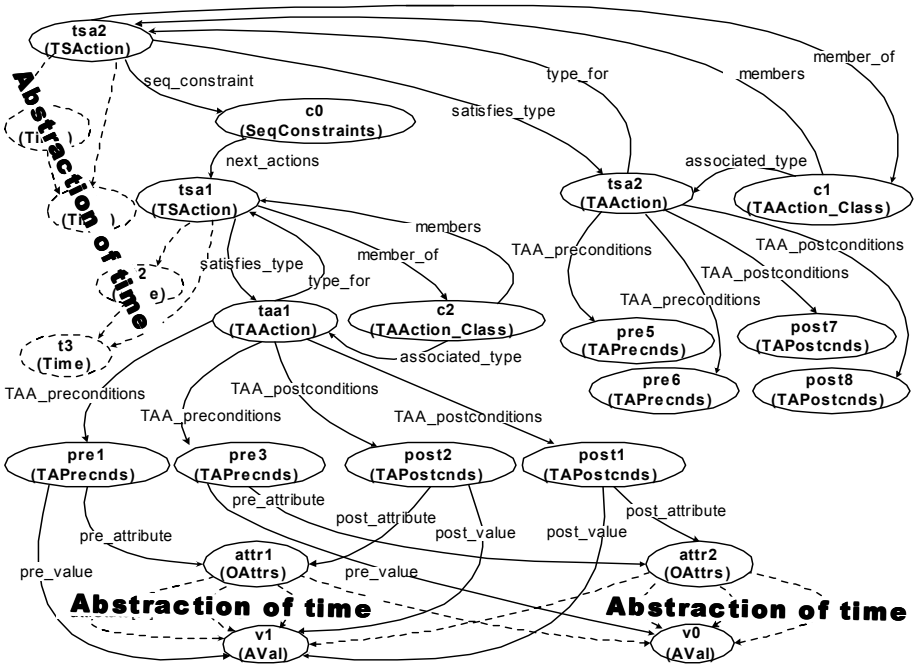



**Figure 7.** Example of the model of a system behavior and state built with Alloy Constraint Analyzer

In the example from figure 7, we suppose that the two attributes (attr1 and attr2) are ordinary attributes (we do not show temporal attributes in this example). To make reading of the model easier, we also do not show relations of TAPreconditions and TAPostconditions with attributes for the *tsa2* action.

The example form figure 7 demonstrates how the abstraction of time can be done: instead of using TSActions (tsa1, tsa2) we can specify TAActions (that are types of TSActions). TAActions specify values for ordinary attributes before and after each TSAction. Thus the model of TAAction does not specify any particular time interval where it may occur and information about actual time intervals can be hidden (we show with dotted lines in figure 7). This allows us to specify infinite behavior.

In order to do this we have to review the definition of TSBehavioralConstraints. We will use a concept of *time abstracted (TA) behavioral constraints*: constraints defined between TAActions. Therefore behavioral constraints of sequentiality define the sequence of TAActions such that this sequence preserves the sequence of

TSActions: if two TSActions in the Time Specific RM-ODP model are sequentially constrained then two corresponding TAActions should also be sequentially constrained. Using TAActions and TA behavioral constraints brings us to the *Time Abstracted RM-ODP model*. To show how Time Abstracted RM-ODP model can be built based on the Time Specific model we use a slightly different example (see figure 8) than the example from figure 7.

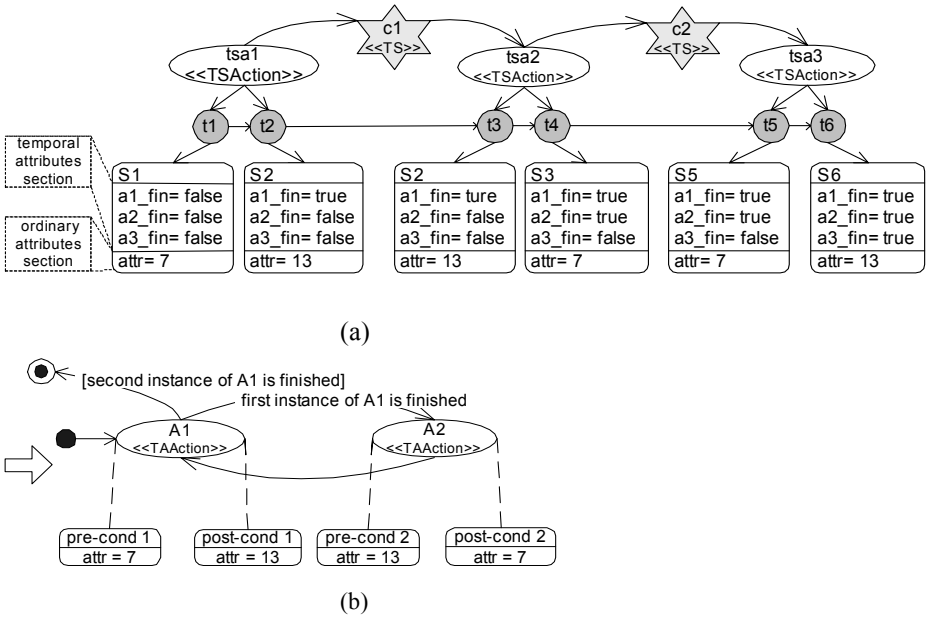



**Figure 8.** RM-ODP diagram: From Time Specific RM-ODP model (8.a) to Time Abstracted RM-ODP model (8.b)

In the example from figure 8.a we suppose that the TSActions tsa1 and tsa3 have the same TA preconditions (attr = 7) and TA postconditions (attr = 13). In that case they can be specified with TAAcion *A1* (see figure 8.b). The TSAction tsa2 has to be specified with another TAAction *A2*.

Time Abstracted RM-ODP model does not include the partially ordered set of time points. It makes it possible to specify the infinite behavior of an object. But still we have to keep the information about the order of TSActions. To keep this information, we have to introduce two elements: initial and final points (black dot and black dot in a white circle). Another thing we should pay attention to is how to specify the constraints of sequentiality between TAActions. Any TAAction in Time Abstracted RM-ODP model may specify several TSActions, such that any TSAction has TSBehavioralConstraints with other TSActions. Thus we have to distinguish between instances of TAActions in order to specify constraints sequentiality correctly. The easiest way to do this is to introduce for each TAAction a counter that shows which instance of this TAAction has been finished. Based on this counter we can specify the sequence of TAActions. In case if some TAAction is followed by several TAActions, we specify conditions at corresponding arrows (for example “Second instance of A1 is finished”). Note that we simplified our notation for the

constraints of sequentiality. We show them as arrows between sequentially constrained TAActions.

## 3.2 Parameterized TAActions

In the previous subsection we saw that by using TAActions we can specify a set of TSActions that assign the same values to ordinary attributes. But what about TSActions that assign different values to ordinary attributes but assign them in a similar way (based on some known mathematical function)? For them we can define TAPostcondition in the following way:

***TAPostcondition (with parameter) [ver1]****: postcondition in the form: equals (attr,val) (or "attr=val"); where val: attr@pre→{values of ordinary attributes} and attr@pre is a value of attr before action.*

Here *val* is a unary function that takes the value of the attribute before action occurrence. TAPreconditions we can keep almost in the same form as before with the difference that *val* becomes a nullary function that can point to any element from the subset of ordinary attribute values:

***TAPrecondition:*** *precondition in the form: equals (attr,val) (or "attr = val"), where attr∈{ordinary attributes} and val:__→{precondition values}⊂{values of ordinary attributes} is an unary function.* In order to simplify our notation we will write these TAPreconditions in the form: "attr∈ {precondition values}".

Figure 9, for example, shows two TSActions (b1 and b2).

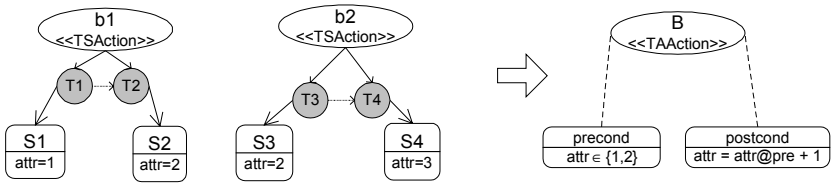



**Figure 9.** RM-ODP diagram: postcondition as a function

In this example we can define the TAAction B with TAPrecondition "attr ∈ {1, 2}" and TAPostcondition "attr = attr@pre + 1". Thus we have defined TAAction with parameterized TAPostconditions. The parameter is the value of an ordinary attribute before the TSAction. In the similar way we can define TAPostcondition that takes TSAction as a parameter. This leads us to the concept of action with a parameter used in many modeling languages. Often a parameter is defined as value that can be passed to the object. For example, UML defines parameter in the following way:

**Parameter** [O1999] *"is an unbound variable that can be changed, passed, or returned. Parameters are used in the specification of operations, messages and events, templates, etc. In the meta-model, a Parameter is a declaration of an argument to be passed to, or returned from, an Operation, a Signal, etc."*

Let's see what parameter means in RM-ODP terms. Figure 10 shows a set of TSActions from the Time Specific RM-ODP model $\{c_0, \ldots, c_N\}$. Only one of them can take place, depending on the choice of environment. Let's suppose that all these

TSActions have similar TAPostconditions: these TAPostconditions differ only in the value that is assigned to the state attribute *attr*.

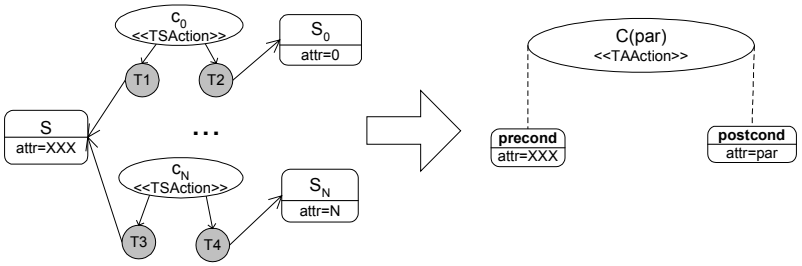



**Figure 10.** RM-ODP diagram: TAAction with parameters

To specify all these TSActions $\{c_0, \ldots, c_N\}$ with one type (one TAAction) we define TAPostcondition with a parameter [version 2]:

***TAPostcondition (with parameters) [ver2]****: postcondition in the form: equals(attr, val) (or "attr = val"), where val: TSActionClass→{values of ordinary attributes} and TSActionClass⊂TSAction.*

In this definition *val* is a unary function that takes as an argument TSAction (from some TSActionClass) and returned a value to be assigned to the attribute *attr*. Based on the definition of TAPostconditions with parameters we can define TAAction (with parameter):

***TAAction (with parameters):*** *It is a type characterizing the set of TSActions: it is a predicate that specifies values of ordinary attributed before and after any TSAction (that is an instance of TAAction). These valued are specified with TAPreconditions and TAPostconditions (with parameter).*

In the example, *C(par)*, par∈ {0...N} in figure 10 is TAAction that characterizes the set of TSActions $\{c_0, \ldots c_N\}$.You can see that we use a parameter *par* in the notation for TAAction. Thus *par* in figure 10 allows us to relate a particular TSAction (the instance of TAAction) with a value assigned to the attribute *attr*.

In this section we considered different TAPreconditions, TAPostconditions and TAActions that have been defined using them. Many other TAActions can be defined by means of mixing the TAPreconditions and TAPostconditions presented in this section. For example, we can specify TAAction with mixed TAPostconditions: we can represent a value that is assigned to an attribute as an n-ary function: *val: $attr_1$@pre, $attr_2$@pre, TSActionClass →{values of ordinary attributes}*. Thus the value assigned to the attribute of an on object depends on: values of two attributes before TSAction and TSAction itself.

# 4 MAPPING RM-ODP SEMANTICS WITH SEMANTICS OF DIFFERENT SPECIFICATION LANGUAGES

The abstraction of time considered in the previous section brings us to a Time Abstracted RM-ODP model. This model can be used as a generic model that we

considered in the introduction. In this section we show how different views can be built on a genetic Time Abstracted RM-ODP model.

We begin with the example of the Time Specific RM-ODP model. We show how this example can be reduced to a Time Abstracted RM-ODP model using TAActions instead of TSActions. Then we consider how three views on the Time Abstracted RM-ODP model can be built. We show the three following view: CCS process algebra view, UML activity diagram view and UML statechart diagram view.

## 4.1 Example

Figure 11 shows the example of a Time Specific RM-ODP model. This model specifies the behavior of an object with nine TSActions. Five of them are TSInternalActions (they take place without the participation of the environment) and four of them are TSInterActions (they take place with the participation of the environment of the object). Names of TSInternalActions start with "a" and names of TSInterActions start with "e".

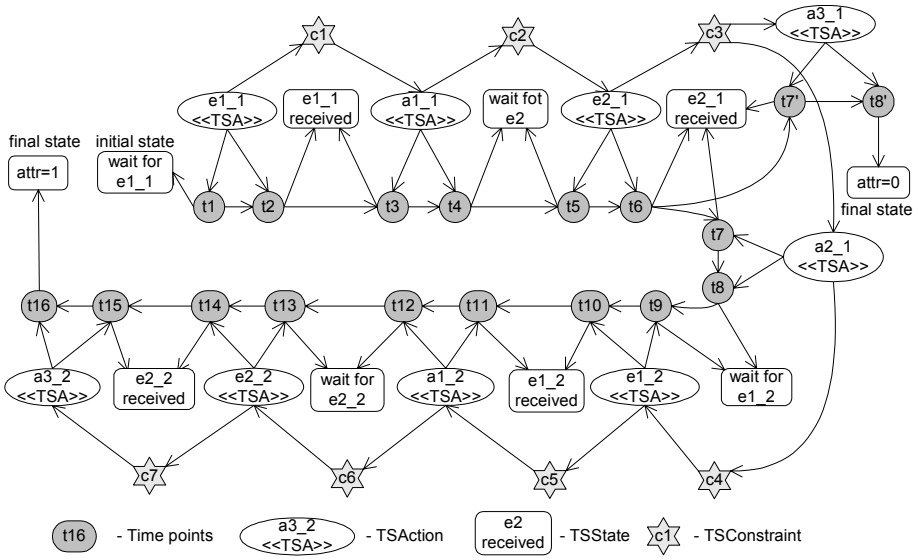



Figure 11. Time Specific RM-ODP model: an example of a behavior

The example shows the system TSStates before (pre-states) and after (post-states) each TSAction, TS Constraints and time points. You can see that a post-state after each TSAction is the same as a pre-state for the next TSAction. However, in general, these pre- and post-states can be different, since some other concurrent process can change the state of a system between two TSActions. Here we suppose that in our system there are no concurrent processes and thus there are no other processes that can change the state of the system between two consecutive TSActions. In this example we suppose that TSActions a1_1 and a1_2; e1_1 and e1_2; e2_1 and e2_2 have the same TA preconditions and TA postconditions. TSActions a3_1 and a3_2 also perform the same functionality (not specified here), with the slight difference

that a3_1 makes the state attribute attr equal to 0, while a3_2 makes this attribute equal to 1.

First, if we make an abstraction of time. This brings us to the following Time Abstracted RM-ODP model:

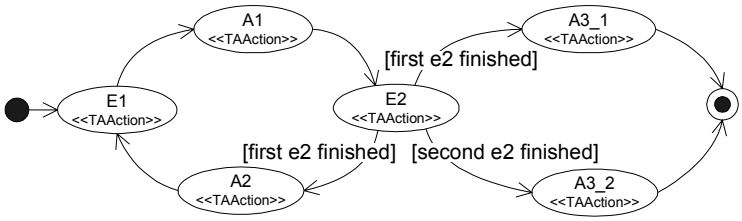
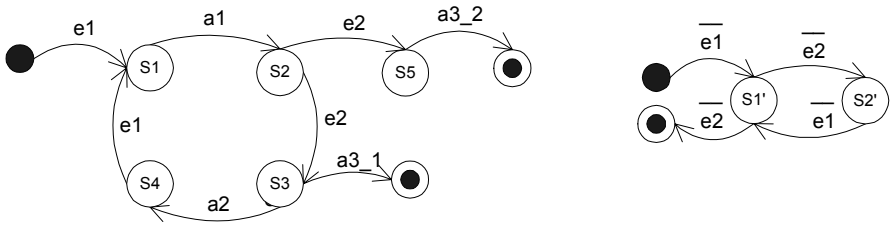



**Figure 12.** Time Abstracted RM-ODP model

Here A1, A2, E1, E2, A3_1 and A3_2 are *TSActions* that characterize the following collections of TSActions from figure 11: {a1_1;a1_2}, {e1_1;e1_2}, {e2_1;e2_2}, {a3_1} and {a3_2} (for the purpose of simplicity we do not show TA preconditions and TA postconditions for these TAActions). Note that we introduced a counter for the action E2 and the conditions on the constraints of the sequentiality. It allows us to specify the same sequence of action instances in figure 12 as the sequence of TSActions in figure 11.

## 4.2 CCS Process algebra

In this section we consider how the Time Abstracted RM-ODP model can be transformed into a CCS [M1999] model. First we explain how to build a CCS transition graph based on the RM-ODP model and then we show a corresponding CCS process expression. A transition graph can be built in the following way: any action becomes arc in the transition graph, constraints of sequentiality become states. Let's note that that just constraints of sequentiality become states in the transition graph but not pre- or post states. Some other concurrent process can change the state of a system between two actions. This means that the pre-state of an action and the post-state of a consecutive action can be different. But constraints of sequentiality in RM-ODP define exactly the same meaning as states in a transition graph: they specify the sequence of actions.




**(a)** **(b)**

**Figure 13.** Transition graphs of the system (a) and its environment (b)

Here also we have to pay attention to the transforming of constraints of non-determinism. In order to express them in the transition graph we have to model action e2 twice. It shows that the system makes the internal choice between the two "branches" of behavior without being influenced by its environment. For a better illustration we also show the behavior model for the system environment. An interaction of an object with its environment can be represented as a reaction between the following pairs of actions and their complements {e1,$\overline{e1}$} {e2,$\overline{e2}$}. The same specification in the form of concurrent process expressions would be:

$System = e1.S1$

$S1 = a1.S2$ $\qquad$ $Environment = \overline{e1}.S1'$

$S2 = e2.S5 + e2.S3$ $\qquad$ $S1' = \overline{e2}.S2' + \overline{e2}$

$S3 = a2.S4 + a3\_1$ $\qquad$ $S2' = \overline{e2}$

$S4 = e1.S1$

$S5 = a3\_2$

Note, that the transition graph in Figure 13 does not allow us to count instances of action e2. Thus e2 in figure 13 can have more than two instances. To have only two instances (e2_1 and e2_2) we have to specify them separately without grouping them into one action.

## 4.3 RM-ODP and UML Statechart and Activity Diagram

The further simplification (using modeling of actions with parameters) of our example from figure 12 leads us to the behavior model shown in figure 14, where the post-condition for action a3(p) is "attr = p". We use the model in figure 14 to show how UML Activity and Statechart views can be defined.

Note, that in all behavior models we considered above, interactions and internal actions are modeled using the same notation (the sign of oval). But UML uses a slightly different notation. UML has the two following terms:

**(UML) Action:** "An action is a specification of an executable statement that forms an abstraction of a computational procedure that results in a change in the state of the model, and can be realized by sending a message to an object or modifying a link or a value of an attribute" [O1999].

**(UML) Event**: "An event is a noteworthy occurrence. For practical purposes in state diagrams, it is an occurrence that may trigger a state transition" [O1999].

Although there is no direct mapping of an RM-ODP interaction and an RM-ODP internal action with a UML Action and UML Event, in our particular example we can conclude that E1 and E2 correspond to UML events and A1, A2, A3 correspond to UML actions. "An event is something done to the object; an action is something that the object does" [S2000]. An event in UML is considered as an action trigger and modeled in the way it is shown in figures 15 and 16.

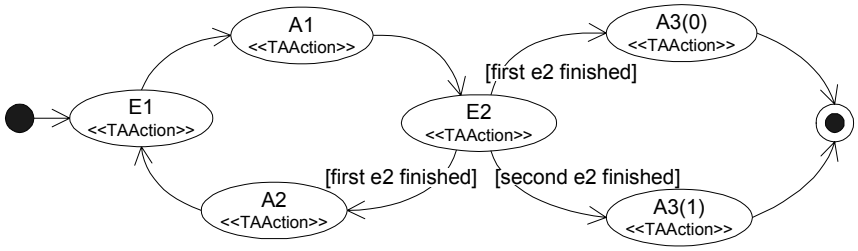



**Figure 14.** RM-ODP diagram: Simplification of the model using actions with parameters

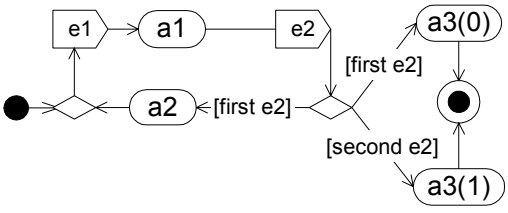



**Figure 15.** UML activity diagram

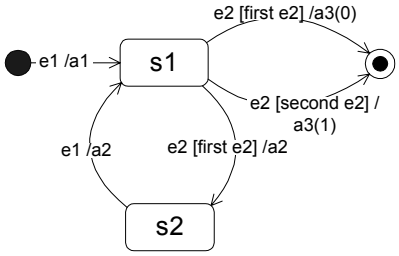



**Figure 16.** UML statechart diagram

# 5 CONCLUSION

In this work we analyzed the possibility of using RM-ODP Part 2 "Foundations of the Open Distributed Processing" as a generic semantic domain for systems behavior modeling. We have considered the minimum set of RM-ODP concepts that a system designer needs for "any kind of modeling activity" [I1996]. These concepts form the generic semantic domain for system behavior modeling and allow a system designer to specify generic behavior models.

RM-ODP behavior models are based on the concept of Time Specific Action (TSAction) and Time Specific State (TSState). Time Specific Actions directly represent things that happen in the Universe of Discourse with explicit reference to time. An object in each time point is specified with one Time Specific State. We call a model that use TSActions and TSStates, a Time Specific RM-ODP model. However, "the incorporation of concrete timing properties leads to a considerable loss of abstractness" [B1991]. To make Time Specific RM-ODP models more abstract and to be able to specify the infinite behavior, we considered a Time Abstracted RM-ODP model. A Time Abstracted RM-ODP model makes an abstraction of time by means of using Time Abstracted Actions (TAActions) and Parameterized TAActions. TAAction characterizes the set of TSActions that assign the same value to some ordinary attributes of an object. Parameterized TAAction characterizes the set of TSActions whose postconditions can be specified as a mathematical function.

We believe that a Time Abstracted RM-ODP model can be used as a generic behavior model. Having a generic behavior model allows a system designer to define different views on this model, where each view addresses particular problems that a system designer wants to solve. Each view may have its specification language. In this work we considered how a Time Abstracted RM-ODP generic model can be seen from the three views done with the following specification languages: CCS process algebra, UML Activity Diagram and UML Statechart Diagram. We explained the mapping of corresponding concepts from the semantic domains of these three languages and from the generic semantic domain based on RM-ODP.

This work continues the work done by Naumenko [N2001] that formalizes the semantics of RM-ODP. The main contribution of this work is the formal definition of TAAction. We show a formal relation of Time Specific Behavior with Time Abstracted Behavior. This relation can be used in case tools to check the consistency between behavior instance diagrams (like UML Sequence Diagram) and behavior type diagrams (like UML Activity diagrams). The definition of Time Abstracted Action is based on the definition of State Structure and Composite State. These concepts extend the notion of composition presented in RM-ODP. RM-ODP defines the composition of object and the composition of behavior. However RM-ODP does not define how the state of the composite object can be defined. In this work we define the Composite State that can be used to specify a state of the composite object.

## ENDNOTES

[i] "Alloy is a language for describing structural properties. It offers declaration syntax compatible with graphical object models, and a set-based formula syntax powerful enough to express complex constraints" [J2000]. See also http://sdg.lcs.mit.edu/alloy/.

[ii] **"Location in time:** An interval of arbitrary size in time at which action can occur." [I1996]

[iii] Do not confuse this Alloy invariant with the invariant defined in RM-ODP. We use an Alloy invariant to guaranty the consistency of concepts on the meta-level. This invariant can not become a part of our model, while the RM-ODP invariant is a specification concept. It is a predicate that can be used in a model. In this work we use the concept of invariant only at the meta-level.

[iv] See http://sdg.lcs.mit.edu/alloy/

[v] RM-ODP defines activity in the following way: "**Activity:** A single-headed directed acyclic graph of actions, where occurrence of each action in the graph is made possible by the occurrence of all immediately preceding actions."

[vi] Post-state is a state of an object after the occurrence of an action.

[vii] This again shows the duality of state and behavior. Constraints of sequentiality are dual with the state information that tells which actions are finished (they specify the same thing from the point of view of behavior and state).

[viii] RM-ODP gives the following definition for preconditions and postconditions:
***Precondition:*** *A predicate that a specification requires to be true for an action to occur.*
***Postcondition:*** *A predicate that a specification requires to be true immediately after the occurrence of an action.*

## REFERENCES


[B1991] Broy, M., *Formal treatment of concurrency and time***,** in *Software Engineer's Reference Book*, J. McDermid, Editor. 1991, Oxford: Butterworth-Heinemann,. p. 23/1-23/19.

[I1996] ISO/IEC 10746-1, 3,4 | ITU-T Recommendation X.902, *Open Distributed Processing - Basic Reference Model - Part 2: Foundations* **.** 1995-1996.

[J2000] Jackson, D., *Alloy: A Lightweight Object Modeling Notation,* Technical Report 797, 2000, MIT Laboratory for Computer Science: Cambridge, MA.

[L1990] Lamport, L. and N.A. Lynch, *Distributed Computing: Models and Methods*, in *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. 1990, Elsevier and MIT Press.

[M1999] Milner, R., *Communicating and Mobile Systems: the pi-Calculus*. 1999: Cambridge University Press.

[N2001] Naumenko, A., *et al. A Viewpoint on Formal Foundation of RM-ODP Conceptual Framework*, Technical report No. DSC/2001/040, July 2001, EPFL-DSC ICA.

[O1999] OMG, *Unified Modeling Language Specification, v 1.3*, 1999.

[P1983] Poincaré H, *The value of science***,** Moscow «Science», 1983

[S2000] Stevens, P. and R. Pooley, *Using UML Software Engineering with Objects and Components (Updated Edition)*. Object Technology Series. 2000.

[W2001] Wegmann, A. and A. Naumenko. *Conceptual Modeling of Complex Systems Using an RM-ODP Based Ontology*. in *5th IEEE International Enterprise Distributed Object Computing Conference - EDOC 2001*. 2001. Seattle, ACTION.

[W1990] Wing, J.M., *A Specifier's Introduction to Formal Methods.* IEEE Computer, 1990. **23(9)**: p. 8-24.